TÉCNICO LISBOA

Evaluation of Real Time Operating System in RISC-V

Rui Silva Ferreira

Electrical and Computer Engineering

Supervisor(s): Prof. Paulo Flores
Prof. Rui Policarpo Duarte

June 2023

Abstract

Real-time operating systems play a critical role in improving an embedded system's performance by efficiently managing multiple tasks. The emergence of the RISC-V instruction set architecture as a standard open-source platform offers unique advantages for embedded systems due to its simplicity and modularity. This thesis will study and adapt an open-source real-time operating system on a RISC-V processor that runs on an FPGA. Given that the RISC-V CPU can be attached to a hardware accelerator, this thesis will also study the interaction between the real-time operating system and the accelerator and how the accelerator impacts the system's performance.

Keywords: RISC-V, RTOS, hardware accelerators

Contents

Chapter 1

Introduction

1.1 Motivation

A Real-Time operating system (RTOS) is one of the components that support the execution of applications in a computational system that can improve the performance of that system. RTOS are often designed to meet the timing requirements of real-time systems, therefore, its use is suitable to manage multiple tasks with varying priorities. An RTOS also provides inter-task communication, mutual exclusion, task synchronization, and power modes. These capabilities are crucial for the system's functioning and to manage the system's resources.

Recently, RISC-V has been establishing itself as a standard open-source instruction set architecture (ISA) both in the academic environment as well as in industry implementations [1]. The open-source nature of RISC-V enables a broader and larger community around it that contributes to its development. Its modular and simplistic approach makes it not only easier but also more suitable for systems with resource constraints.

Hardware accelerators are used in embedded systems by adding custom digital circuits to speed-up algorithms that usually are the computational bottleneck. When using hardware accelerators to perform computationally intensive tasks for which they were specifically designed the system is able to better utilize its resources. The use of accelerators can also improve the responsiveness of the system to the time constraints of real-time systems.

In addition to the performance increase that a hardware accelerator provides, the use of an RTOS in such systems will help take advantage of the CPU time that would otherwise be wasted. Using RISC-V based processor will allow the developer to implement a system that is specific for its application, optimizing, even more, the resource utilization of the system.

1.2 Objectives

The primary objective of this thesis is to incorporate a RTOS into a RISC-V processor inside an SoC programmed in a field programmable gate array (FPGA), and use the RTOS capabilities to manage and

use a hardware accelerator. The initial work is to survey the existing SoC based on RISC-V implementations and RTOS. After choosing a RISC-V core and system on chip (SoC), they will be targeted on an FPGA. The third stage is to implement and evaluate how the RTOS performs in the RISC-V Soc. The following action is to include a hardware accelerator in the system. Finally, the last stage is to evaluate the performance of the system and to observe how the use of an accelerator will impact the performance of the RTOS.

1.3 Document Outline

The next chapter, Chapter 2, provides an overview of the RISC-V instruction set architecture and a comparison between RISC-V SoCs.

Chapter 3 provides an overview of the inner mechanisms of real-time operating systems, going more in dept to mechanisms that will be important to this thesis work such as how RTOS manages threads and switches between contexts as well as how RTOS share resources between the different tasks.

Chapter 4 introduces the work that will be done on this thesis. After that, it presents the first solutions for the integration of hardware accelerators into a RTOS that will be studied, as well as the preliminary work already done to implement the most appropriate RTOS on the chosen hardware. It also includes performance metrics that will be used to evaluate the use of RTOS in the RISC-V based hardware accelerators.

Chapter 5 presents the conclusions taken from this initial work and presents the work plan for the thesis development.

Chapter 2

RISC-V Instruction Set Architecture

2.1 RISC-V Overview

RISC-V [2], [3] is an instruction set architecture (ISA) that is rapidly growing due to the fact that it is an open-source ISA, contrary to other popular ISAs such as ARM's and Intel's x86. The RISC-V ISA is defined to be a base ISA that is restricted to a minimal set of instructions capable of providing a target to compilers, assemblers, operating systems, etc. In addition to the base ISAs, RISC-V International [4] also provides standard extensions and the capability to add custom extensions that enable the developer to add additional capabilities for specific applications.

There are four base ISAs that are characterized by the width of the integer registers and by the number of integer registers. The two primary base ISAs are RV32I and RV64I which provide 32-bit and 64-bit address space, respectively. Both have 32 integer registers. The RV32E is a subset of the RV32I that only has 16 registers and was created to support small microcontrollers. The fourth base ISA is RV128I and supports a 128-bit address space. As mentioned before, RISC-V International provides a few extensions that add some functionalities to the base instruction sets. The most significant among them is the 'M' standard extension that adds integer multiplication and division for signed and unsigned integers. Other extensions are the 'A' standard extension that adds atomic instructions, the 'F' standard extension that provides floating-point operations, floating-point status and control registers and floating-point registers and the 'Zicsr' extension that allows access to the control and status register, this enables interrupt and exception handling.

RISC-V also supports privileged levels and extensions for the privileged ISAs. There are currently three privilege levels: User (U), Supervisor (S) and Machine (M). The privilege modes exist to add protection between components of the software stack. The Machine mode has the most privileges and runs implementation specific firmware. The Supervisor mode is typically used by operating systems and it gives access to most of the hardware. The mode that offers the least privileges is the User mode. It's in this mode that user applications generally run. All implementations must provide the Machine mode since this is the only privileged mode that has access to the entire hardware.

2.2 RISC-V Hardware Implementations

There are many hardware implementations of RISC-V, however, for the purposes of this work, the only implementations analyzed were the ones listed on the list [5] under the SoC platforms section, since the focus of this thesis is not to develop the SoC itself. Even though there are several implementations listed it's necessary that the chosen SoC is under an open source license, has good documentation, and an active community support.

Three candidate RISC-V implementations were selected to be studied in detail: LiteX [6], SweRVolf [7] and NEORV32 [8]. Table 2.1 presents a comparison of the main key factors, such as the base ISA, the supported extensions, the provided RAM size, the system clock speed, the supported simulation tool, and the quality of the documentation provided.

	LiteX	SweRVolf	NEORV32
Base ISA	RV32I	RV32I	RV32I
RISC-V Extensions	[M][A][F][D][C]	[M][C]	[M][C][Zicsr][Zicntr]
RAM	256MB	128MB	configurable
system clock	1MHz	50MHz	150MHz
Simulation tool	Verilator	Verilator	ISIM
Support for RTOS	Zephyr	Zephyr, TockOS	Zephyr, FreeRTOS
Documentation	decent	good	decent

Table 2.1: Comparison between RISC-V SoC

Looking at table 2.1 it's possible to see that the NEORV32 SoC is capable of higher clock speeds and has support for two of more relevant RTOS as will be discussed in Chapter 3. However, none of these features is enough to make it the obvious choice, since it also is the more complex one. The LiteX SoC has native support for the Zephyr OS, having a guide on how to install and simulate the RTOS on the Zephyr documentation. The SweRVolf SoC has good support and documentation and already supports relevant RTOS, Zephyr and TockOS. SweRVolf SoC is the SoC selected for this thesis work as it offers resources for the integration of the RTOS that reduces the development effort and ensures a correct integration of the chosen RTOS.

2.3 Conclusion

This chapter provided an overview of the RISC-V ISA. It explained the four base ISA, RV32I, RV64I, RV32E and RV128 as well as some of the more relevant standard extensions. It also introduced the three privilege levels of RISC-V, machine, standard and user. They provide different levels of access to the machine, adding protection between components of the software stack.

A survey of the existing RISC-V hardware implementations was conducted, emphasizing SoC platforms. Of the ones listed in [5], the SoC that were studied in more detail were LiteX, SweRVolf and NEORV32. These implementations were selected based on their open-source license, documentation

and community support. Finally, the chosen implementation is SweRVolf as it provides sufficient resources for the installation of the RTOS and offers good documentation and installation guides that will greatly aid the work conducted in this thesis.

Chapter 3

RTOS Background

3.1 RTOS Overview

A Real-time System is a system where its correctness does not depend only on its output but also depends on the time that the output takes to produce. These systems can be divided into three types: hard real-time systems; soft real-time systems; and firm real-time systems. Hard RTS are those where the system will fail if any task does not finish in its set time. In soft RTS, tasks can fail their time constraints, however, the performance of the overall system can be reduced because of this. Firm RTS have tasks that need to be processed before their deadline, but having some tasks fail that deadline will not make the system fail.

In more complex cases, the use of Operating Systems (OS) is necessary to manage the resources to meet the demands of the system. A Real-time Operating System is an OS designed to take into consideration the necessity of RTS to comply with time constraints. Since an RTOS is an OS it must have some basic functionalities in its kernel such as a process scheduler, memory management, a file system, tasks, multitasking, synchronization, timers and clocks, inter task communication, I/O, and memory management.

One of the most important capabilities of an OS is its capability to multitask. Since only one task per CPU core can run at any given time, the OS can swap between tasks to improve the system's performance. In those cases, the OS saves the context of the initial task and starts running the highest priority task. Multitasking is especially important in RTOS. When a high priority task is ready to execute it is imperative that the RTOS is able to execute that task so that no time constraint is broken.

In typical RTOS, tasks are grouped into three states: running, ready, and blocked. The running task is the task that is executing on the CPU. Ready tasks are those tasks that are ready to run and are waiting to go to the running state. Finally, the blocked state includes the tasks that cannot run at that moment, usually because they are waiting for an event to occur. When there are multiple tasks ready to be executed the kernel must decide which task is going to use the available resources. In this case, the kernel uses a scheduler. The scheduler is the process that controls which task is running on the CPU at any given time. To execute the tasks, the scheduler makes use of threads. Threads are parts

of processes that can run independently of other threads since they have their own stack and registers copy. Threads within the same process also share some resources such as memory and files.

In RTOS, several tasks have to wait for events provided either by other tasks or by the hardware in the system. These events are called interrupts. To handle the interrupts, the RTOS makes use of a mechanism called interrupt service routine (ISR).

3.2 Task Scheduling and Context Switching

The scheduler is the mechanism that is in charge of choosing the task that is executed at any time. The scheduler is especially important in RTOS since it has to ensure that all tasks meet their time constraints. Scheduling algorithms can be divided into two types, preemptive, where the scheduler has the capability to switch the running task even if that task does not go into the blocked state, and cooperative, where the scheduler does not swap the task that is in the running state, but instead, the running task yields the resources periodically. There are a few situations where the scheduler needs to decide what task should run next: when the running task finishes, when the running task moves to the blocked state or when an interrupt happens. In RTOS the most commonly used scheduling algorithms are preemptive, since they provide more control over what task is in the running state.

There are a number of preemptive scheduling algorithms. The most commonly used is priority scheduling or highest priority first. In this scheduling algorithm, every task is assigned a priority. The scheduler then makes sure that the task with the highest priority is always running. If a low priority task is running and a new high priority task goes to the ready state, then that new task starts running and the old task goes to the ready state. Another scheduling algorithms are first come first served, or FIFO, where the selected task is the first task to arrive at the ready state, and shortest job first, where the task with the least amount of computing left executes first. One example of a cooperative scheduling algorithm is round-robin scheduling. With this algorithm the tasks are not assigned any priority, however, the tasks are swapped after a set period of time. It's very common for RTOS to use this type of scheduler to assign running time to tasks of the same priority in priority scheduling.

Another important mechanism of the RTOS is how it handles context switching. Context switching is the mechanism that saves the current state of the task for later use, usually to its stack, and initializes the state of the new running task. This adds overhead that, for RTSs, can make the system fail. That is why it's important that the context switching in RTOS takes as little time as possible. The frequency of context switches depends on the scheduling policy and preemptive scheduling, usually, has more context switches than non-preemptive schedulers. This means that the scheduler has to weigh the context switching delay when choosing what task to execute.

3.3 Interrupt Handling

One of the ways that tasks go from the blocked state to the ready state is when some event occurs. This event can come in the form of software interrupts, such as messages from other tasks, and hard-

ware interrupts, such as input from a keyboard or mouse. To handle the interrupts there is a mechanism called Interrupt Service Routine (ISR). The ISR is a high priority task that runs every time an interrupt occurs and manages the cause of that interrupt. After the triggering of an interrupt, an interrupt request (IRQ) that is associated with the interrupt signals the ISR. The ISR then utilizes an interrupt vector table to determine the correct interrupt handler function. All of this adds a delay which is called interrupt latency. More precisely, interrupt latency is the time between the interrupt signal and the conclusion of the ISR.

Before starting the ISR, the CPU has to save the context of the task it was executing. And after completing the interrupt handling routine the CPU can restore the state it was on or start a new task that was triggered by the interrupt. Sometimes the running thread may need to prevent ISRs from executing, due to some time-sensitive task or critical section operation. In those cases it is possible to disable the ISR, however, this should only be done when no other solution is available, since a more critical task may be delayed because of it.

3.4 Resource Sharing

Since the purpose of the RTOS is to manage and run multiple processes, it's important to mention how the RTOS shares the data and the resources used between the tasks. It's generally unsafe for systems capable of multitasking to access the same data without some kind of protection. Protection may be achieved using the following methods.

3.4.1 Temporarily Masking

Temporarily masking is when the user temporarily disables interrupts and the ability for the kernel to switch tasks. This way the task can process the data knowing that no other tasks will change that data before the processing finishes. Temporarily masking is the fastest way to make sure that the current task has exclusive access to the systems resources, however it should only be used when the longest path through the protected section is smaller then the interrupt latency since this will prevent any other task from running, even higher priority ones.

3.4.2 Semaphores and Mutexes

A semaphore is an object that controls the access to a specific resource. The semaphore stores how many tasks can access that resource and how many tasks are accessing it at that time. When the number of tasks reaches the limit, no other tasks are allowed to access that resource. This way whenever a task needs to access shared resources it increments the semaphore counter. When that thread finishes it's processing it decreases the semaphore counter. When only one task can access that resource the semaphore is called a binary semaphore.

Similarly to binary semaphores, there is a mechanism called mutex. A mutex ensures mutually exclusive access to a hardware or software resource. When a task wants to access a mutually exclusive

resource it must start by locking the associated mutex. After the task is finished it unlocks the locked mutex. While the mutex is locked no other tasks can execute because they cannot access the resources protected by the mutex. The task only starts executing again when the mutex is unlocked. One key difference between mutexes and semaphores is priority inversion. Priority inversion happens when a low priority task locks a mutex and a high priority task is blocked by that mutex. When this happens the priority level of the original task is elevated so that the higher priority task does not have to wait for the low priority task to be assigned CPU time. One problem that comes with mutexes is deadlocking. This happens when, for example, task A locks task B, however, before locking, task B also locked task A. When this happens both tasks are unable to continue executing. This can be prevented by carefully designing the program.

3.4.3 Message Passing

Message passing mechanisms such as FIFOs, stacks, message queues and a mailbox can be used to share data between tasks without the use of shared memory. With this mechanisms a task only shares the data after its processing is done, ensuring that the other tasks only start processing after receiving the data from the first task. Message passing can also make use of priority inversion when a high priority task needs data from a low priority task. Deadlock can also occur when some task is working on data needed for another task, but needs data from the blocked task to continue.

3.5 RTOS Comparison

Before starting the incorporation of the accelerator into the RTOS it's necessary to choose a reliable RTOS. A series of open source RTOS, seen in [9], were analyzed. Immediately, several of the RTOS in the list were excluded because they either stopped being updated or because they lacked good documentation. After this initial filtering, the RTOS that stood out from the rest were: FreeRTOS [10], RIOT [11] and Zephyr [12].

3.5.1 FreeRTOS

FreeRTOS is an RTOS that targets small embedded systems. The RTOS has a minimal footprint and prioritizes its small memory size, low overhead, and fast execution. The RTOS Kernel can be tailored to the specific use case of the system with a configuration file called FreeRTOSConfig.h. In addition, its small and simple kernel makes it ideal for systems with limited resources.

Besides the expected capabilities of an RTOS, FreeRTOS also provides some more specific ones. First, it has a very small footprint needing only 4.4 KB of ROM and 500 Bytes of RAM. FreeRTOS has a very fast context switch delay, needing only 84 cycles to swap between tasks. The RTOS uses a highest priority scheduling algorithm, with Round Robin time slicing. Both of these algorithms can be disabled in the configuration file. On top of all this, FreeRTOS also provides excellent documentation and vast community support.

3.5.2 RIOT

RIOT is an RTOS designed to cater to the requirements of Internet of Things devices. The main goal of RIOT is to provide a lightweight but versatile RTOS that runs on a low-power microcontroller but can also use the full power of larger devices with more resources. To achieve this, RIOT uses a modular architecture that allows developers to include only the components needed for the requirements of the system. The programming model is based on event-driven programming. This allows for a highly responsive and efficient system, that still supports multi-tasking. RIOT has a very small footprint requiring only 3.2 KB of ROM and 2.8 KB of RAM. It also has a very small interrupt latency that takes only 50 clock cycles. The RTOS uses a highest priority first scheduling algorithm and a first come first served algorithm for tasks with the same priority.

3.5.3 Zephyr

Zephyr is an RTOS developed by the Linux Foundation. It offers a scalable and secure platform for IoT devices. Similarly to RIOT, it offers a modular architecture that allows for developers to only select the necessary features of the RTOS. This RTOS provides developers a feature rich software that can run in a footprint as small as 8kB and as large as to reach megabytes and supports both 32bit and 64bit architectures.

Besides the expected capabilities of an RTOS, Zephyr offers additional capabilities. First, it has a small footprint, requiring 8 KB of ROM and 4 KB of RAM. It also has a fast context switch delay, taking 264 clock cycles. This RTOS uses a highest priority first scheduler and utilizes the algorithm longest waiting first for tasks of equal priority. It also allows the developer to choose if RTOS will use any time slicing or not. It is also possible to define the task queuing strategies used in the scheduler.

3.5.4 RTOS Feature Comparison

There are a few characteristics to compare between the RTOS. First, it is important to make sure that the selected RTOS is supported in the chosen SoC, and if not, if the board fits the installation requirements, such as ROM and RAM. It is also important to study how fast can the RTOS process an interrupt and how fast can it switch its context. Finally, it's important to choose an RTOS that has good documentation and strong community support. Table 3.1 summarizes the characteristics mentioned. Note that some of the more precise metrics, such as interrupt latency and context switch delay, do not have any strict value since they depend on the use case of the RTOS.

	FreeRTOS	RIOT	Zephyr
Minimum ROM size	4.4 KB	3.2 KB	8 KB
Minimum RAM size	500 Bytes	2.8 KB	5 KB
Interrupt Latency	—	50 cycles	—
Context Switch Delay	84 cycles	—	264 cycles
Scheduling Policy	Highest Priority	Highest Priority	Highest Priority
Time Slicing Policy	Round Robin	none	configurable
Documentation	Excellent	Good	Excellent

Table 3.1: Comparison between FreeRTOS, RIOT and Zephyr

Looking at the metrics mentioned in Table 3.1, t it's possible to see that FreeRTOS should perform better for smaller systems that have less RAM and ROM available. It is also a better RTOS when dealing with small tasks and the context switch delay is more impactful. At the same time, due to its vast variety of capabilities, it's possible to conclude that ZephyrOS should perform better for larger systems. In this system the context switch delay should not be as impactful, making the pros of the RTOS outweigh the cons. The support for the chosen SoC is also an important feature to consider. None of the RTOS analyzed have direct support in their documentation page, however, both FreeRTOS and Zephyr have online guides on how to use them with the chosen SoC, SweRVolfX.

Since ZephyrOS is the RTOS with better support and documentation it will be the one used. However, it's important to keep in mind that, if the capabilities provided are not useful or even worsen the performance of the overall system, the lighter option should be used.

3.6 Conclusion

This Chapter provides an overview of RTOS. It explains the different types of RTS: hard, soft, and firm real-time systems. It also explains how the performance of those systems varies with the correctness of the time constraints. After that, it summarizes how the RTOS manages the different tasks making use of the scheduler and context switching. It also describes some of the most common preemptive scheduling algorithms: priority scheduling, first come first served, and shortest job first. After that, it explains how an RTOS makes use of the ISR to handle interrupts. The last mechanisms of a RTOS summarized in this Chapter are the ones used for resource sharing. This includes temporarily masking, Semaphores, Mutexes, and message-passing mechanisms.

Finally, this chapter provides a comparison between three RTOS: FreeRTOS, RIOT and Zephyr. To select the RTOS used for this thesis some characteristics of the RTOS were analyzed. First, there are some more technical characteristics that are important such as the minimum ROM and RAM size that they need, the interrupt latency, the context switch delay, and the scheduling policies used. Since all the RTOS provide the necessary capabilities for this thesis work, the decision was made based on the documentation and community support that each RTOS provides. In this field, ZephyrOS is the best candidate.

Chapter 4

Preliminary Evaluation of RTOS for RISC-V Based Hardware Accelerators

4.1 Introduction

The purpose of this thesis is to study and evaluate the impact the use of a hardware accelerator has on an RTOS running on an SoC with a RISC-V based processor. This Chapter presents two initial solutions on how to manage a hardware accelerator with an RTOS. After that, it describes the experiments done to better understand the mechanisms of an RTOS. The first experiment was the development of a basic program that utilizes FreeRTOS capability to multitask. The developed program was then flashed into the board Raspberry Pi Pico. The second experiment was building the SoC SweRVolf using the Vivado Block Design Tool [13]. The last experiment was to run Zephyr in a simulation of the synthesized SoC.

The initial work done will help better understand how to design and build an SoC, making it easier for any future modification to the SoC. The installation of the two RTOS will help study the necessary procedures to incorporate a hardware accelerator into the RTOS. Finally, the use of already established hardware, such as the Raspberry Pi Pico board, helps to focus on the study of the RTOS mechanisms, without worrying about the correct use of the hardware.

4.2 Integrating a Hardware Accelerator in a RTOS

Typically, when using hardware accelerators to improve processing performance, the CPU has to wait for the accelerator to finish it's job before continuing with it's execution. This results in a waste of processing time in the CPU since it will stay waiting for the accelerator to finish its work. One way to improve this is to implement a system that uses the CPU's resources while the accelerator is performing its work. The use of a RTOS should make this management much easier.

A first naive solution to be explored is to make use of the accelerator as if it was a function to be

executed by the tasks. The communication between the task and the accelerator is managed by the task itself. This would mean a direct communication path, however, the task itself would have to make sure that the accelerator is not in use. If the hardware accelerator is already in use, the task would need a way to know when it is free. When the task manages to start using the accelerator it can free the CPU and allow other tasks to execute. In this solution, the accelerator will have to wake the sleeping task by triggering an event or interrupt. This interrupt is then perceived by the ISR that, in turn, will change the state of the task that needs the data from the hardware accelerator. This also means that the system will need to keep track of which task is waiting for the accelerator. Figure 4.1 shows a block diagram with the interaction of one task with the accelerator.

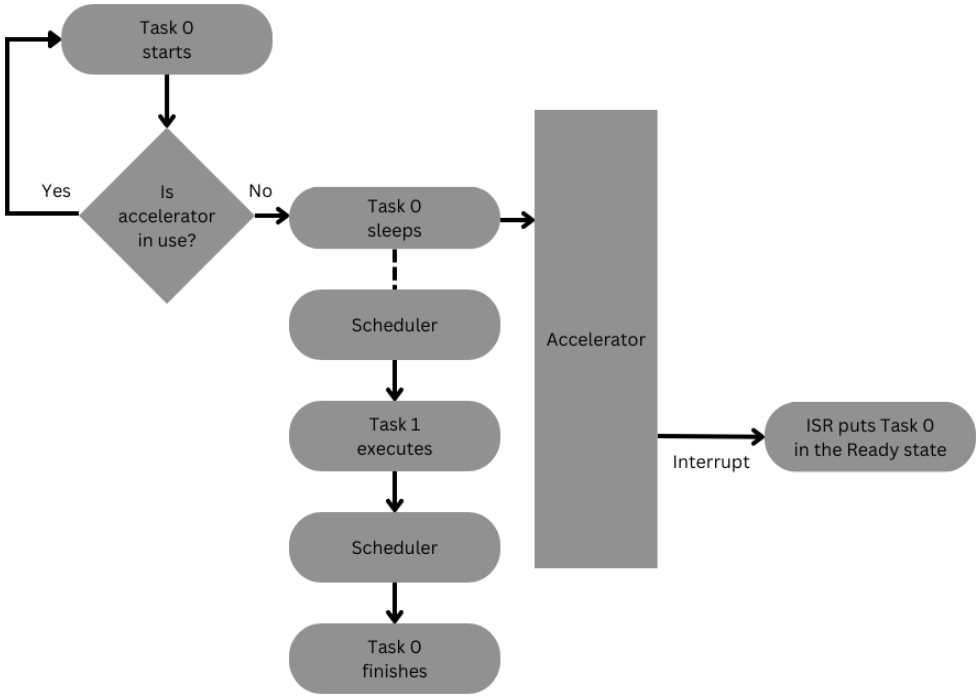

Figure 4.1: Block diagram of a first naive implementation

Another solution is to have a specific task that manages the hardware accelerator. This would mean that anytime a task needs to use the accelerator, it must communicate with the task that controls it. The controller task can keep track of all the tasks that need the accelerator and manage its use. This will help reduce what the tasks need to do when they have to use the accelerator. However, this also adds overhead to the communication between the running task and the accelerator. The overhead exists because the task now needs to communicate with the controller task in addition to the communication between the controller task and the accelerator. Eventually, when the accelerator finished its work, the controller task can send the output to the original task. Figure 4.2 shows a block diagram with the interaction of one task with the task that manages the accelerator.

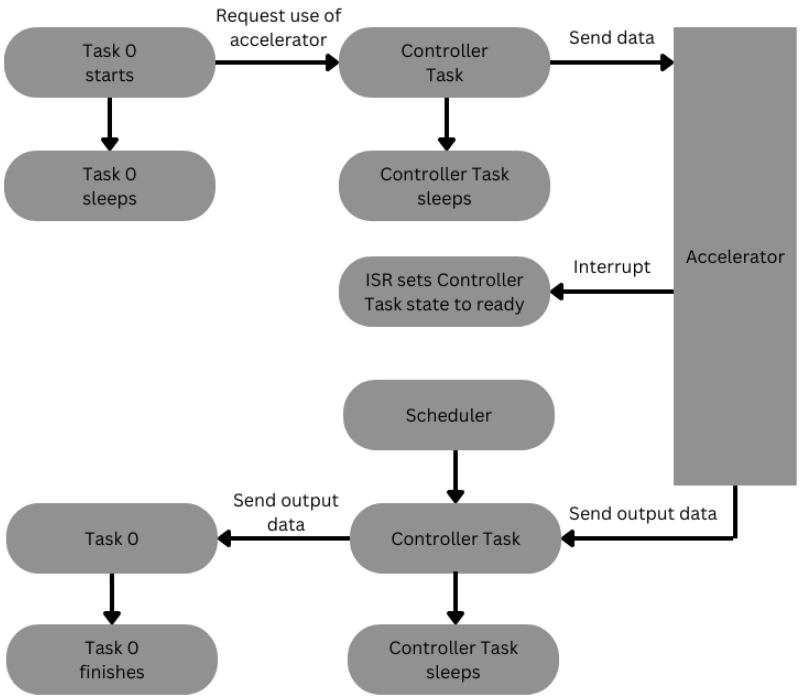

Figure 4.2: Block diagram of a second naive implementation

4.3 Preliminary Work

The programs used and developed in this section can be found in the github repository [14].

4.3.1 FreeRTOS on Raspberry Pi Pico

The main objective of flashing a FreeRTOS program on the Raspberry Pi Pico is to understand the basic procedures necessary to implement custom programs. Since the Pi Pico is a commercial hardware platform, the documentation and support for the board are vast. The vast documentation also allows for a better understanding of RTOS mechanisms and how to use them, without needing to worry about configuring the RISC-V on the FPGA..

Before running any example programs, it was necessary to install the toolchain for the Pi Pico board. The Pi Pico toolchain allows developers to use the VSCode IDE with CMake to build the developed programs and generate the binary files. CMake is a tool that allows programmers to easily create binary files. The developer uses the CMakeLists.txt file to tell CMake how the program is structured and what libraries are necessary. After building the binary files, it is possible to flash the device and execute the programs.

The first two example programs to run were Blinky and Hello World. The Blinky example is a basic example that flashes the board LED repeatedly. The Hello World example is a basic example that prints

”Hello, World!” to the serial monitor repeatedly. These examples demonstrate important capabilities of embedded systems: how to use GPIO, and how to use the serial monitor. The use of GPIO is essential for connecting extra modules to a system. The use of a serial monitor is essential for debugging and communicating information from the system to the user.

Finally, to execute programs with FreeRTOS the following steps were taken. First, create a new project with the FreeRTOS source code in the project folder. Second, add the path to the necessary libraries to the CMakeLists.txt file. These libraries include the file that ports FreeRTOS to the board and the file with the memory allocation instructions. The last step is to create the source files of the programs. The created program was a combination of the two examples programs already described. This program uses different tasks to blink the LED and to print ”Hello, World!” to the serial monitor. Both of the tasks use timers that trigger the ISR when it’s time to return to that task. The ISR then tells the scheduler that the task is ready to execute. The scheduler switches the execution context to the ready task. The task does its job and sleeps for a predefined time, waiting to execute when the timer triggers a new interrupt. In Figure 4.3, it is possible to see the Pi Pico flashing the LED and, in the background, the serial monitor with ”Hello, World” being printed repeatedly.

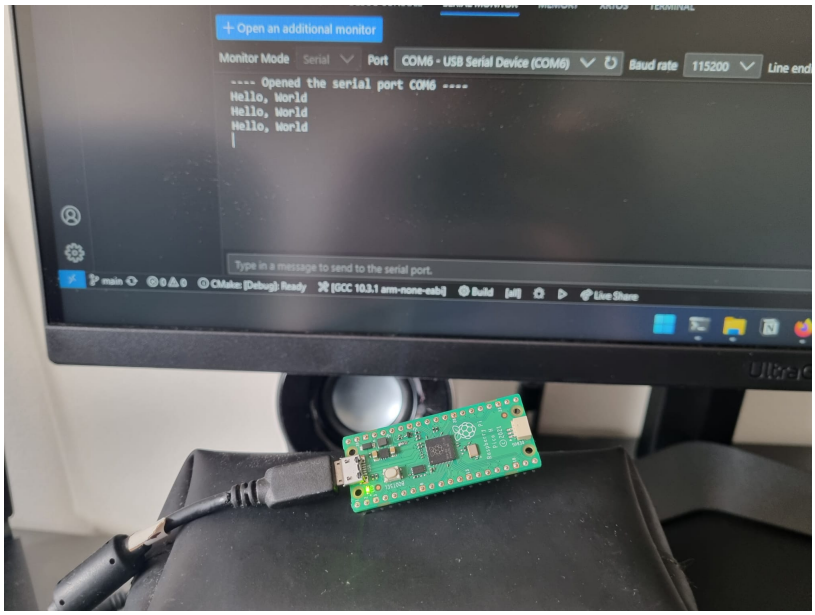

Figure 4.3: FreeRTOS executing on RaspberryPi Pico

4.3.2 SweRVolf SoC Synthesis

To better understand how to build an SoC a subset of the SweRVolf was built using basic building blocks in the Vivado Block Design Tool. The SoC built is divided into three main major blocks: the SweRV Core, an Interconnect block (AXI Interconnect, AXI Wishbone Bridge, and Wishbone interconnect), and peripherals (System controller, Boot-ROM, and GPIO). Figure 4.4 shows a block diagram of the SoC to be implemented.

The first step to build the SoC is to create a project targeting the board Nexys A7-100T using Vivado.

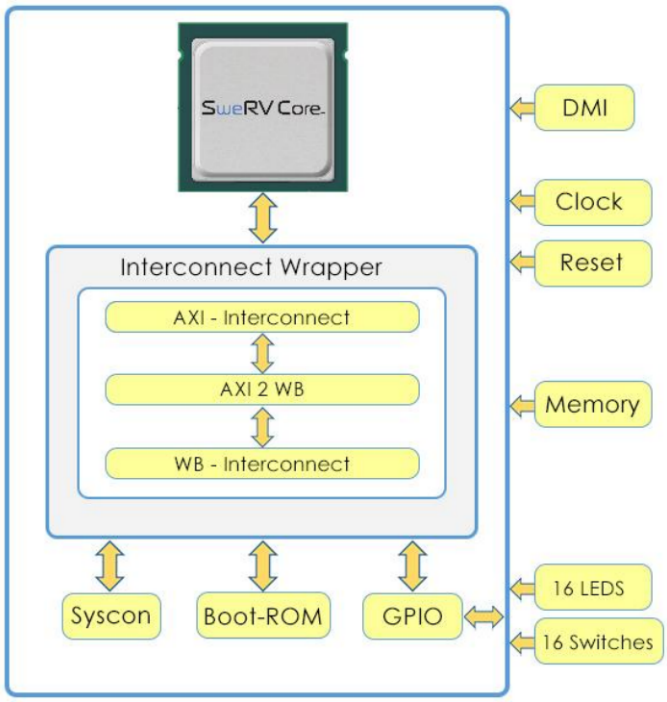

Figure 4.4: SweRVolf block diagram

It's necessary to add all the individual modules to the sources of the project. These modules will map the connections so that the block design generated by Vivado can utilize the board components, for example, the LEDs and the switches. The second step is to create the block design and add the necessary modules: the SweRV Core wrapper module; the interconnect wrapper module, which contains the three interconnect modules; the Boot-ROM module; the GPIO module; the Syscon wrapper module that contains the system controller; and 32 bidirect modules, that will be attached to the GPIO and contain the 16 LEDs and 16 switches. The third step is to connect the different modules and add the external connections for the I/O pins. The external connections include the RAM, the clock, the reset, the debug module interface, and the bidirect GPIO modules. The last step is to synthesize the SoC and generate the bitstream file that will be used to configure the FPGA.

4.3.3 Execution of Example Programs on the SweRVolf SoC

Using PlatformIO [15] and a binary file generated by Verilator [16] it is possible to build programs and simulate their execution on the SoC. Verilator uses the block design created in Vivado to generate the binary file for the simulation. It is now possible to create a PlatformIO project that will use the bitstream file created by Vivado and the binary file created by Verilator. The first PlatformIO project is a simple RISC-V assembly file that initializes the register t3 to 15 and subtracts 1 in every iteration of a loop until it reaches 0. Using the tool GTKWave it's possible to see the execution of the simple instructions.

To run more complex programs with Zephyr it's necessary to install FuseSoC [17]. FuseSoC is a package manager and a build system for HDL code. In this case, FuseSoC is the tool used to build the SweRVolf SoC and install Zephyr. With the full environment installed two example programs were

executed. The ”hello world” program is a standard program, used to ensure that the system is correctly built. After running the program with FuseSoC the expected output was printed to the terminal.

The final program executed was the dining philosophers problem. The philosophers problem is an example problem that illustrates synchronization issues and how to solve them. This problem consists of five philosophers sitting at a round table with a fork in between each of them. The philosopher can be in one of three states: thinking, eating, or starving. To eat, a philosopher needs both forks on each side of him. The problem starts when two philosophers need the same fork to eat. This can create deadlocks. To solve this problem, the example program uses the Dijkstra's solution. The Dijkstra's solution introduces a ”butler”, which essentially provides a way to keep track of the available forks around the table. When a philosopher wants to eat, he requests the butler and when the forks are available the butler gives them to the philosopher. When the philosopher finishes eating, he signals the butler, who, in turn, assigns the forks to the next philosopher. This program correlates to the main problem of this thesis, which is how to best assign limited resources and when to put tasks to sleep while waiting for those resources.

4.4 Conclusion

This Chapter introduces two solutions that will be explored in this thesis. In the first solution, the tasks communicate directly with the hardware accelerator. This solution has a smaller communication path between the task and the accelerator. However, this solution also increases the complexity of the tasks waiting for the accelerator. The second solution uses only one task to handle the communication with the accelerator. With a task controlling the communications, it is possible to use a queue to manage who needs to use the accelerator. The use of this task will add overhead between the communications, but reduce the complexity of each task requesting the use of the accelerator.

The second section of this Chapter reports the preliminary work done to better understand the inner mechanisms of an RTOS and the necessary procedures to build an SoC. The first experiment conducted was to use a commercial embedded system to execute an RTOS and develop skills in the use of tasks. The second experiment provided insight into how to design an SoC. This will facilitate the addition of different modules to the SoC. This Chapter also describes the steps taken for the use of simulators to execute Zephyr in the SweRVolf SoC. One of the programs executed was the philosophers problem. This problem illustrates how to manage a limited resource using the different mechanisms of an RTOS.

Chapter 5

Conclusions

The simplistic approach of the RISC-V ISA paired with the use of RTOS enables developers of embedded systems to build optimized systems for their specific application. However, with an increase in the volume and processing requirement of data, further improvements must be done to the embedded systems that process that data. One improvement is the incorporation of hardware accelerators.

In this initial work, an overview of the SoC and RTOS mechanism necessary is performed. From the vast variety of existing SoC, the SweRVolf is chosen as a base to build the full system. SweRVolf was chosen due to its complete documentation, community support, and support for popular RTOS. The RTOS chosen for this system is Zephyr since its modularity and vast documentation allow the creation of a system that is optimized for this thesis application.

The preliminary work described in Chapter 4 presents the necessary tools to build a complete system. Utilizing already established hardware to study the RTOS basic mechanisms allowed for a further understanding of such mechanisms without the interference of possible hardware debugging. Designing a basic SoC similar to the one chosen provided knowledge on how to later add the hardware accelerator to the SoC. Finally, the use of simulation tools to build a Zephyr application targeting the designed SoC provided a basic understanding of the necessary procedures to build the final system.

The integration of an RTOS into a RISC-V based hardware accelerator requires an evaluation and analysis of performance metrics. These metrics should help to study the efficiency and resource utilization of the system. Some of the key performance metrics are: ROM and RAM utilized, FPGA area utilized, context switching delay, interrupt latency, message passing delay, and overall speedup of the application.

5.1 Work Plan

Following the work already done, the next step will be to build the full SoC with the hardware accelerator. With a full SoC, the second step will be to implement the two solutions provided in Chapter 4. Solution one is for every task to communicate with the accelerator separately, reducing the communication time but increasing the complexity of the tasks. Solution two is to have a central task that manages

the accesses to the accelerator, adding one step to the communication process, but reducing the complexity of the tasks. After that, the study of new and improved solutions will be done. The last step is the evaluation of the developed solutions. The timeline of the work schedule can be seen in Figure 5.1.

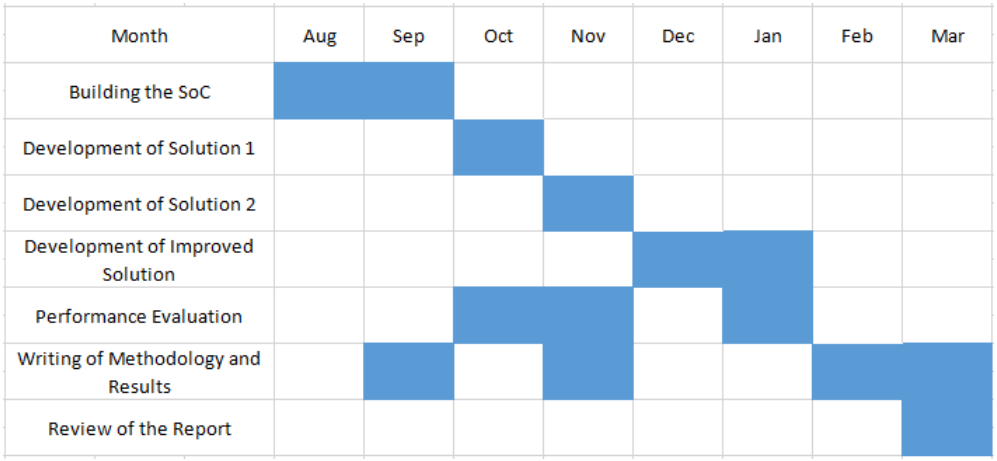

Figure 5.1: Work Plan

References

[1] Samuel Greengard. Will risc-v revolutionize computing? *Commun. ACM*, 63(5):30–32, apr 2020. ISSN 0001-0782. doi: 10.1145/3386377. URL https://doi.org/10.1145/3386377.

[2] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V International, 20191213 edition, December 2019.

[3] Krste Asanovic Andrew Waterman and John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. RISC-V International, 20211203 edition, December 2021.

[4] Risc-v international. https://riscv.org/, . Accessed: 2023-04-15.

[5] Risc-v cores and soc overview. https://github.com/riscvarchive/riscv-cores-list, . Accessed: 2023-04-15.

[6] Litex documentation. https://github.com/enjoy-digital/litex. Accessed: 2023-05-06.

[7] Swervolf documentation. https://github.com/chipsalliance/Cores-SweRVolf. Accessed: 2023-05-06.

[8] The neorv32 risc-v processor. https://github.com/stnolting/neorv32. Accessed: 2023-05-06.

[9] List of open source real-time operating systems. https://www.osrtos.com/. Accessed: 2023-05-28.

[10] Freertos: Real-time operating system for microcontrollers. https://freertos.org/. Accessed: 2023-05-28.

[11] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6): 4428–4440, 2018. doi: 10.1109/JIOT.2018.2815038.

[12] Zephyr project, a linux foundation project. https://www.zephyrproject.org/. Accessed: 2023-05-28.

[13] Vivado block design tool overview. https://www.xilinx.com/products/design-tools/vivado.html. Accessed: 2023-06-10.

[14] Github repository with preliminary work. https://github.com/Rui-Ferreira3/thesis-repo.git. Accessed: 2023-06-10.

[15] Platformio. https://platformio.org/. Accessed: 2023-06-10.

[16] Verilator user's guide. https://verilator.org/guide/latest/. Accessed: 2023-06-10.

[17] Fusesoc reference manual. https://fusesoc.readthedocs.io/en/stable/ref/index.html. Accessed: 2023-06-10.